A Study of Element Fill-in-Blank Problems for C Programming Learning Assistant System

Htoo Htoo Sandi Kyaw, Nobuo Funabiki, Shune Lae Aung, Nem Khan Dim, and Wen-Chung Kao

***Abstract*—Nowadays, *C programming* is essential for university students to study various programming languages, algorithms, and computer architecture. Previously, we have developed *Java programming learning assistant system (JPLAS)* for studying *Java programming*. *JPLAS* provides several types of exercise problems to cover different study stages, including the *element fill-in-blank problem (EFP)*. An EFP instance asks students to fill in the blank elements in the given source code. The correctness of the answer is marked through *string matching.* In this paper, we study the EFP for *C programming learning assistant system (CPLAS)*, by extending our works for JPLAS. To automatically generate a feasible EFP instance, the *graph-based blank element selection algorithm* is newly designed and implemented for C programming. For evaluations, we generate 19 EFP instances using C source codes for basic grammar concepts, and fundamental data structures and algorithms, and assign them to 42 students in a Myanmar university. The solving results confirm the effectiveness of EFP in detecting the students who may have difficulty in studying C programming and the hard topics for them.**

***Index Terms*—C programming, self-study, CPLAS, element fill-in-blank problem, algorithm, graph.**

I. Introduction

Nowadays, *C programming* has still been considered to be the most fundamental programming language. A lot of universities around the world are teaching C programming, or its object-oriented extension C++, as the first computer programming, not only in IT departments but also in other departments such as mechanical engineering and electrical engineering. Besides, students in IT departments should study C programming in parallel with computer architecture courses, because it needs, for example, to study the access to memories or registers for efficient programming on the given computer architecture. As a result, C is selected as the third most popular programming language despite the age since the appearance [1].

Previously, to assist *Java programming* educations in universities, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* [2]-[4]. Java has been widely used in IT societies as the practical object-oriented programming language. Thus, Java programming has been educated in many IT departments. JPLAS provides several types of exercise problems to cover different study levels of Java programming studies. Then, for any exercise problem, the answer from a student is marked automatically using the software in JPLAS, to support self-studies of Java programming.

For the first learning stage, JPLAS offers the *element fill-in-blank problem (EFP)*. The EFP is designed for novice students to study Java grammar and basic programming skills through *code reading*. In the EFP instance, a Java source code that has several blank elements is given to students, where the blanks are shown explicitly in the code. Then, the students are requested to fill in the blanks by typing the correct elements. Here, an *element* represents the least unit in the code, which includes a *reserved word*, an *identifier*, and a *control symbol*. The correctness of the answer from a student is verified through *string matching* with the corresponding original element in the code.

In the EFP instance, the original element in the source code for any blank must be the unique correct answer, to avoid confusions among novice students. Thus, we proposed the graph-based *blank element selection algorithm* to select blank elements automatically. This algorithm can select as many blanks as possible that have grammatically correct and unique answer elements in the source code.

JPLAS has successfully been applied to students in several universities in Japan, Myanmar, and Indonesia, where a lot of EFP instances have been generated to be solved. Then, it will be a good idea to develop a *C Programming Learning Assistant System (CPLAS)* by extending the works on JPLAS to C programming in order to advance C programming educations.

In this paper, we propose the *element fill-in-blank problem (EFP)* as the first step of CPLAS. To automatically generate an EFP instance for C, the *blank element selection algorithm* is newly designed and implemented for C programming where the conditions for selecting blank elements are redefined. This algorithm consists of the four steps: 1) it analyzes the C source code using the original *parser*, 2) it generates a *constraint graph* by selecting each candidate blank element in the code as a *vertex* and connecting any pair of vertices by an *edge* such that their incident elements cannot be blanked simultaneously for unique correct answers, 3) it derives the *compatibility graph* by taking the complement of the constraint graph, and 4) it seeks a *maximal clique* of the compatibility graph to find a maximal set of blank elements with unique answers.

In the evaluations, we first verify the correctness of the algorithm through applications to various C source codes. The uniqueness of the correct answer for any blank was manually confirmed. Then, we generate EFP instances for C programs by applying the algorithm and assign them to university students in Myanmar. The results confirm the

Manuscript received October 20, 2020; revised February 6, 2021.

H. H. S. Kyaw and N. Funabiki are with Okayama University, Okayama, Japan (e-mail: pxs93q36@s.okayama-u.ac.jp, funabiki@okayama-u.ac.jp).

S. L. Aung and N. K. Dim are with University of Yangon, Yangon, Myanmar (e-mail: shunelaeaung@gmail.com, nemkdim@gmail.com).

W.-C. Kao is with the Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan (e-mail: jungkao@ntnu.edu.tw).

doi: 10.18178/ijiet.2021.11.6.1520

effectiveness of the proposal in detecting the students who may have difficulty in studying C programming and the hard topics for them.

The rest of this paper is organized as follows: Section II introduces various related works. Section III overviews the element fill-in-blank problem in CPLAS. Section IV proposed the refined blank element selection algorithm for C programs. Section V evaluates the EFP in CPLAS. Finally, Section VI concludes this paper with future work.

II. Related Works

A lot of work has been reported for C programming educations. In this section, we introduce related works to the fill-in-blank selection algorithm.

In [5], Kashihara *et al*. proposed a method of blanking an important point of data or control flow in a C code using *Program Dependence Graph (PDG)*, to make instructive fill-in-blank problems without considering semantic aspects. PDG can represent the relationship of data dependency and control flows between commands using a graph. By solving the problem, students can improve skills of processing the flow of the program.

In [6], Terada *et al*. proposed a methodology to automatically generate fill-in-blank problems for C codes. To automatically generate problems, two key constituents, the selection of exemplary code and the selection of places to be blanked, are presented. For the first one, they use k-means clustering with silhouette analysis is used to select exemplary code from the Aizu Online Judge system (AOJ) which has over 3 million of source codes. For the later one, a model based on a bidirectional *Long Short-Term Memory Network (Bi-LSTM)* with a sequential *Conditional Random Field (CRF)* is used. The blank selection methodology selects the important places in the process flow of the source code and creates blanks there. In the future, we will consider the use of process flow of the source code in our study.

In [7], Brusilovsky *et al*. developed the *QuizPACK* system that can generate parameterized exercises for the C language and automatically evaluate the correctness of student answers by comparing them to the correct ones provided by the teacher. The fill-in-blank problem in *QuizPACK* asks the values of particular variables in a source code and is similar to the *value trace problem* in [3]. On the other hand, the fill-in-blank problem in this paper requests filling in the blank elements that are composed of reserved words, identifiers, or control symbols for basic grammar and code reading studies.

In [8], Kakeshita *et al*. developed a programming education support tool called *Pgtracer*. *Pgtracer* utilizes fill-in-the-blank questions composed of a source code and a trace table. The blanks in the code and the trace table must be filled by the students to improve the code reading while solving the questions. In *Pgtracer*, they are manually selected by the teacher. On the other hand, the blanks are automatically selected using the *blank element selection algorithm* in our proposal.

In [9], Barros *et al*. developed an e-learning tool called *ProPAT*. This tool is implemented as an *Eclipse* plug-in with two perspectives: the *teacher perspective* and the *student perspective*. This tool allows students in the first computer science course to learn how to program using *pedagogical patterns*, which are the set of programming patterns recommended by computer science educators. For answer marking, the tool includes a program diagnosis system that uses *Model Based Diagnosis* techniques. The difficulty of the tool is that a teacher needs to collect the *programming patterns*. In our proposal, a teacher only needs to select source codes.

III. Overview of Element Fill-in-Blank Problem for C

In this section, we present the element definition in EFP for C programming, the example instance, and the parser for the blank element selection algorithm.

A. Element Definition in EFP for C

An *element* in EFP for C programming is defined as the least unit of a source code. In an EFP instance, a reserved word, an identifier, a conditional operator, a memory operator, a preprocessor, and a control symbol can be blanked among the elements in the code, if it gives the unique answer.

A *reserved word* signifies a fixed sequence of characters that has been defined in C grammar to represent a specific function. It is expected that students should master the proper use in learning programming. An *identifier* is a sequence of characters defined in the code by the author to represent a *variable* or a *function*. A *conditional operator* is used in a conditional statement to determine the state. A *memory operator* is used as the pointer to a variable * or to get the address of the variable &. A *preprocessor* includes #, *include*, *define*, <, >, **.**, and *h* for a *header file*. A *control symbol* in the paper indicates other grammar element, including ., :, ; , ( ), and { }.

B. Example EFP Instance

code 1 shows an example EFP instance. This instance is generated from **code2**. On the blank elements, # at _1_ is the preprocessor directive, **.** at _2_ and *h* at _3_ are for the header file extension, *main* at _4_ is the identifier, *int* at _5_ is the reserved word for data type, ; at _6_ is the control symbol, *printf* at _7_ is the identifier, ; at _8_ is the control symbol, & at _9_ is the memory operator, *number* at _10_ is the identifier, and *return* at _11_ is the reserved word.

code 1

```
1   _1_ include<stdio _2_ _3_ >
2   int _4_ ()
3   {
4      _5_ number _6_
5      _7_ ("Enter an integer:") _8_
6      scanf ("%d", _9_ _10_);
7      printf("You entered: %d", number);
8      _11_ 0;
9   }
```

code 2

```
1   #include<stdio.h>
2   int main()
3   {
```

```
4       int number;
5       printf("Enter an integer:");
6       scanf ("%d", &number);
7       printf("You entered: %d", number);
8       return 0;
9   }
```

C. Parser

To generate an EFP instance, the source code needs to be separated into a collection of elements with the associated attributes. In this study, the *C parser* is implemented to achieve it.

Fig. 1 shows the interaction between the *lexical analyzer* and the *parser*. The *parser* is also called the *syntax analyzer*. The *lexical analyzer* transforms a given C source code into a sequence of *lexical units* or *tokens* that represent the least elements to compose the code. It can classify each element into either a reserved word, an identifier, a symbol, or an immediate data. The output of the lexical analyzer becomes the input to the *parser* for the syntax analysis.

In this study, we originally implement the *lexical analyzer* for a C source code and adopt *CUP* [10] for the *syntax analyzer*. *CUP* is an open-source system for generating LALR parsers based on the grammar for which a parser is needed. The output of *CUP* includes *sym.java* and *parser.java* classes. The *sym.java* class contains a series of *constant declarations* for the *symbol table*, and the parser.java class includes a parser itself. Table I shows the associated information contained for each symbol in the *symbol table*. In the implementation of the blank element selection algorithm, the *tokens* from the *lexical analyzer* are used in the vertex generation, and their associated symbol information are used in the edge generation.

TABLE I: SYMBOL INFORMATION

item	content
symbol	symbol of element
line	row index of element
column	column index of element
count	number of element appearances
order	appearing order of element in the code
group	statement group index partitioned by { and }
depth	number of { from top

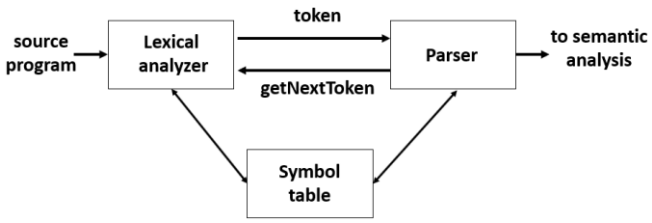

Fig. 1. Interaction between lexical analyzer and the parser.

IV. BLANK ELEMENT SELECTION ALGORITHM

In this section, we propose the blank element selection algorithm for C programming.

A. Algorithm Overview

In the algorithm, first, the *constraint graph* is generated from the given code, where a *vertex* represents a candidate blank element, and an *edge* does the constraint such that their incident elements cannot be blanked simultaneously for unique correct answers. Second, the *compatibility graph* is derived by taking the complement of the constraint graph. Last, a *maximal clique* of the compatibility graph is sought to obtain a maximal set of blank elements with unique answers.

B. Vertex Generation for Constraint Graph

In the constraint graph, each vertex represents a candidate element for being blank. The candidate elements or vertices are extracted by applying the parser to the source code.

C. Edge Generation for Constraint Graph

Each edge is generated between any pair of two vertices or elements that should not be blanked at the same time. There are three categories to represent the constraints in selecting blank elements with unique answers.

1) Group Selection Category: In the *group selection category*, all the elements related with each other in the code are grouped together. There are five conditions of this category. To elaborate, we use the following simple code.

code 3

```
1   #include<stdio.h>
2   float sample_method(int p1)
3   {
4     float tax = 1.08f;
5     retrun p1 * tax;
6   }
7   int main()
8   {
9     int var1 = 10;
10    float var2 = sample_method(var1);
11    printf("indata= % d", var1, "outdata= % d",          var2);
12    return 0;
13  }
```

a) Identifier appearing two or more times in the code

The multiple elements representing the same identifier that has the same scope are grouped together. A *scope* indicates the range in the code where a variable or a function is referred using the same name or identifier [11]. If all such elements are blanked, any student cannot answer the original identifier. For example, in **code 3**, *var1* appears three times with the same scope at lines 9, 10, and 11, which belong to this condition.

b) Pairing reserved words composed of three or more elements

The three or more elements representing the reserved words in pairs are grouped together. If all of them are blanked, the unique answers may become very difficult or impossible to answer. Besides, one of those elements could be a hint to derive the other element for novice students such as the following case:

- switch-case-default

c) Data type for variables in equation

The elements representing the data types for the variables in one equation are grouped together. For example, in *sum* = *a* + *b*, the data types of the three variables, *sum*, *a*, and *b*, must be the same. If a variable is casted like *sum = (int) a + b*, the casted data type *int* is also included in the group.

d) Data type for method and its returning variable

The elements representing the data type of a function and its returning variable are grouped together. For example, in **code 3**, *float* at lines 2 and 4 are grouped.

e) Data type for arguments in method

The elements representing the data type of an argument in a function and its substituting variable are grouped together. For example, in **code 3**, *int* at lines 2 and 9 belong to this condition through line 10.

The data type in (c) – (e) must be the same if at least one element in these groups is overlapped. Thus, after every group is found, the groups from (c) – (e) that contain an overlapped element are merged into one group. In each group, one vertex is randomly selected first. Then, edges are generated between this vertex and the other vertices so that at least this selected element is not selected for blank.

2) Pair Selection Category: In the *pair selection category*, the elements appearing in the code in pairs are grouped together so that at least one element from each pair is not selected for blank. For each pair, an edge is simply generated between the two corresponding vertices. Four conditions of this category are illustrated as follows.

a) Elements appearing continuously in statement

The two elements appearing continuously in the same statement in the code are paired. If both of them are blanked, their unique correct answers may not be guaranteed, or may become a significant challenge for novice students. For example, in **code 3**, *float* and *sample_method* at line 2 are paired. If this condition is removed, the following problem can be generated from line 2 in **code 3**, which will provide novice students with an exceedingly difficult demanding job:

2: _1_ _2_ _3_ _4_ _5_)

b) Variables in equation

The elements representing any pair of the variables in an equation are paired. If both are blanked, the unique answers become impossible because the reversed order is also grammatically correct. For example, for *sum = a + b*, *sum = b + a* is also feasible. If three or more variables are included in an equation, any pair of combinations can be found here.

c) Pairing reserved words

The two elements are paired to represent the pairing reserved words. If both are blanked, the unique correct answers may not be guaranteed, or may put a heavy burden on novice students. Besides, one of those elements can be a hint to derive the other one, including the following two paring reserved words:

- *if-else*
- *do-while*

d) Pairing control symbols

The two elements representing a pair of control symbols, namely ( ) (bracket) and { } (curly bracket), are paired. Even if both are blanked at the same time, the code can be grammatically correct. Furthermore, novice students are expected to carefully check them in their codes to avoid making mistakes. For example, in **code 3**, { } at lines 3 and 6 are paired.

3) Prohibition Category: In the *prohibition category*, an element is prohibited from the blank selection because it does not satisfy the uniqueness with the high probability. There are three conditions for this category.

a) Identifier appearing only once in code

The selected element representing the identifier in this category appears only once in the code. If it is blanked, a student cannot answer the original identifier.

b) Operator

The element representing an operator such as the arithmetic operator: =, +, -, *, /, the comparative operator: <, >, <=, >=, ==, !=, and the logical operator: &, |, ^, ! is selected to this category. If an operator is blanked, a student cannot answer the original one unless the proper explanation on the specification related to the operator is given. For example, in **code 3**, * at line 5 is prohibited.

c) Constant

The element representing a constant is selected to this category. If it is blanked, a student cannot answer the original constant. For example, in **code 3**, 10 at line 9 is prohibited.

D. Example Constraint Graph

Fig. 2 illustrates the constraint graph for *sample_method* in **code3**. A *broken line* signifies that the two incident elements are grouped together by the *group selection category*, where the associated number represents the selecting condition in the category. For example, two *p1* are connected by condition (1) and two *float* are by (4). A *straight line* signifies that they are grouped together by the *pair selection category*. For example, *float* and *sample_method* are connected by (1), *p1* and *tax* are by (2), (and) are by (4). Moreover, a *broken circle* represents an element in the *prohibition category* that must not be selected as a blank element, where = and * are prohibited by (2), and *1.08f* is by (3).

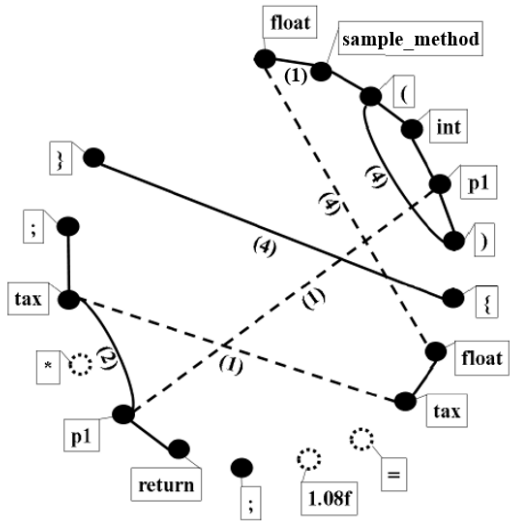

Fig. 2. Constraint graph for *sample_method* in **code 3**.

E. Compatibility Graph Generation

By taking the complement of the *constraint graph*, the *compatibility graph* is generated to represent the pairs of elements that can be blanked simultaneously. Fig. 3 illustrates the generated compatibility graph for *sample_method* in **code 3.**

F. Maximal Clique Extraction for Compatibility Graph

Finally, a *maximal clique* of the *compatibility graph* is extracted by a simple greedy algorithm to find the maximal number of blank elements with unique answers. A *clique* of a graph represents its subgraph where any pair of two vertices is connected by an edge. The procedure for our algorithm is described as follows:

1) Calculate the *degree* (= number of incident edges) of

every vertex in the *compatibility graph*.

2) Select one vertex among the vertices whose degree is maximum. If two or more vertices have the same maximum degree, select one randomly.
3) If the selected vertex is a *control symbol* and the number of selected *control symbols* exceeds 1/3 of the total number of selected vertices, remove this vertex from the *compatibility graph* and go to (5). This step is introduced to avoid too many blanked control symbols.
4) Add the selected vertex for blank and remove it as well as its non-adjacent vertices from the *compatibility graph*.
5) If the *compatibility graph* becomes null, terminate the procedure. Otherwise, go to (2).

TABLE II: EFP INSTANCES FOR BASIC GRAMMAR CONCEPT

problem ID (PID)	basic grammar concept	number of lines	number of blanks
1	standard I/O	8	15
2	function, while loop	28	15
3	recursion	17	18
4	file I/O	17	14
5	function	19	16
6	nested loop	23	20
7	pointer	18	17
8	function	24	20
9	conditional statement	15	14

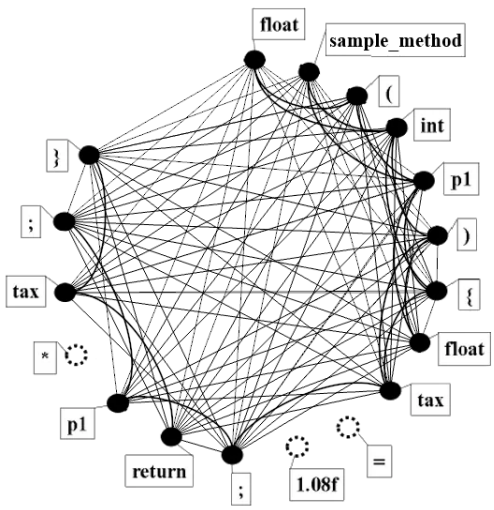

Fig. 3. Compatibility graph for *sample_method* in **code 3**.

V. EVALUATION

In this section, we evaluate the *element fill-in-blank problem (EFP)* in studying C programming by students through applications to 42 students in a Myanmar university who have studied C programming for at least one year.

A. EFP for Basic Grammar Concepts

First, we evaluate the EFP using source codes for basic grammar concepts for C programming.

1) Generated EFP Instances: We generated nine EFP instances for studying basic grammar concepts for C programming using source codes in [12]. Table II shows the problem ID (PID), the grammar concept, the number of lines in the source code, and the number of blanks for each EFP instance.

2) Student Correct Answer Rate: First, we analyze the student performances. Table III shows the distribution of the correct answer rates of the students. This table indicates that 38 students among 42 achieved over 90% rate, which suggests most of the students are satisfactory in studying C programming. On the other hand, two students did not reach 80% rate in solving the EFP instances for basic grammar concepts. The teacher will need to care them.

TABLE III: CORRECT ANSWER RATE DISTRIBUTION FOR BASIC GRAMMAR CONCEPTS

range of correct answer rate	number of students
70% - 79%	2
80% - 89%	2
90% - 99%	6
100%	32

TABLE IV : SUBMISSION TIMES DISTRIBUTION FOR BASIC GRAMMAR CONCEPTS

admission times range	number of students
9	3
10 - 25	9
26 - 50	9
51 - 75	5
76 – 100	5
101 – 125	6
126 - 150	2
151 – 175	0
176 – 200	2
201 – 225	1

3) Student Submission Times: Table IV shows the distribution of the number of answer submission times of the students to mark the answers. Three students correctly solved any instance by submitting the answer only once. They understand C programming well, and carefully check their answers before submissions. On the other hand, one student submitted answers more than 200 times. Although this student achieved 100% correct rate, he/she may not well understand C programming. The teacher will need to follow it.

4) Individual EFP Instances: Next, we analyze the solving results of the individual EFP instances by the students. Table V shows the instance ID, the number of students who did not attempt to solve, the total number of answer submissions, and the average correct answer rate among the 42 students.

This table indicates that all the instances achieved over 90% correct rate, which confirms that they are suitable for novice students in studying C programming. However, two students did not attempt to solve the instance with ID=6 for *nested loop* where the correct rate is smallest. The complex structure of the source code using nested loops may be hard for them. We will investigate the improvement for better understanding, which will be in future works.

TABLE V: SOLVING RESULT IN EACH EFP INSTANCE FOR BASIC GRAMMAR CONCEPTS

instance ID	number of unattempted students	number of submissions	average correct rate
1	0	327	99%
2	0	309	98%
3	0	419	97%
4	0	182	98%
5	0	366	98%
6	2	265	93%
7	0	381	98%
8	0	258	98%
9	0	227	97%
average	0.22	303.78	97.33%
SD	0.67	77.46	1.73%

TABLE VI : EFP INSTANCES FOR DATA STRUCTURES AND ALGORITHM

problem ID (PID)	data structures and algorithms	number of lines	number of blanks
1	Stack	54	35
2	Queue	73	46
3	Hash table	102	45
4	Heap data structure	57	32
5	Binary tree	47	15
6	Breadth first search algorithm	98	46
7	Depth first search algorithm	61	26
8	Quick sort	38	19
9	Longest common sequence	38	23
10	Dijkstra algorithm	48	24

B. EFP for Data Structures and Algorithms

Next, we evaluate the EFP using C source codes for data structures and algorithms.

1) Generated EFP Instances: We generated 10 EFP instances using C programming source codes for data structures and algorithms in [13]. [14]. Table VI shows the problem ID, the data structure/algorithm, the number of lines in the source code, and the number of blanks for each EFP instance. When compared with the instances in Table II, those in Table VI has more numbers of lines and blanks, which suggests they are more difficult.

TABLE VII: CORRECT ANSWER RATE DISTRIBUTION FOR DATA STRUCTURES AND ALGORITHMS

range of correct answer rate	number of students
70% - 79%	1
80% - 89%	0
90% - 99%	15
100%	5

TABLE VIII : SUBMISSION TIMES DISTRIBUTION FOR DATA STRUCTURES AND ALGORITHMS

admission times range	number of students
10	0
11 - 40	9
41 - 80	4
81 – 120	3
121 – 160	2
161 – 200	0
201 – 240	0
241 – 360	3

2) Student Correct Answer Rate: Unfortunately, this time, only 21 students among 42 solved the EFP instances for data structures and algorithms. We need to investigate why the remaining students did not solve them and how we encourage them, which will be in our future works.

Table VII shows the distribution of the correct answer rates of the students. This table indicates that 20 students among 21 achieved over 90% correct rate. This student percentage is almost similar to the previous result.

3) Student Submission Times: Table VIII shows the distribution of the number of answer submission times of the students. Here, any student did not achieve 100% rate by submitting the answer only once to each instance. One student achieved 100% rate with the minimum submission times of 16. Three students submitted answers more than 250 times for 10 instances. Although two students achieved 98% correct rate and one did 92% rate, they may not well understand C programming for data structures and algorithms.

4) Individual EFP Instances: Finally, we analyze the solving results of the students in the individual EFP instances for data structures and algorithms. Table IX shows the instance ID, the number of students who did not attempt to solve, the total number of answer submissions, and the average correct answer rate among the 21 students.

This table shows that any EFP instance achieved over 90% correct rate and can be suitable for novice students. Only one student did not attempt to solve the instance with ID=5 for *binary tree*, where the correct rate is smallest among the 10 instances. However, these instances for data structures and algorithms can be too difficult for the students who did not solve them. We will follow up the students to solve them.

TABLE IX: SOLVING RESULT IN EACH EFP INSTANCE FOR DATA STRUCTURES AND ALGORITHMS

instance ID	number of unattempted students	number of submissions	average correct rate
1	0	224	99%
2	0	493	96%
3	0	270	94%
4	0	176	95%
5	1	63	93%
6	0	131	96%
7	0	64	97%
8	0	103	96%
9	0	184	94%
10	0	114	96%
average	0.1	182.2	95.6%
SD	0.31	128.15	1.71%

VI. CONCLUSION

This paper studied the *element fill-in-blank problem (EFP)}* for *C programming learning assistant system (CPLAS)*. An EFP instance asks students to fill in the blank elements in the given source code, where the correctness of the answer is marked through *string matching*. To automatically generate a new EFP instance from a source code, the *blank element selection algorithm* is newly designed and implemented for C programming, by redefining the conditions for selecting blank elements. For evaluations, 9 EFP instances of basic grammar concepts and 10 EFP instances of data structures and algorithms were generated and solved by 42 and 21 students, respectively. Their solving results confirmed the effectiveness of EFP in detecting the students who may have difficulty in studying C programming and the hard topics for them. In future works, we will investigate improvements of EFP for better understanding and higher motivations of students, generate various EFP instances to cover broad topics, and evaluate the effectiveness in studying C programming through applications to students in different universities.

CONFLICT OF INTEREST

"The authors declare no conflict of interest".

AUTHOR CONTRIBUTIONS

Htoo Htoo Sandi Kyaw designed and implemented the proposal, conducted the experiments, and wrote the paper as the main author. Nobuo Funabiki gave the idea of the proposal and supervised the whole activities including the experiments and the paper writing. Shune Lae Aung and Nem Khan Dim assigned the problems to their students and collected the data. Wen-Chung Kao advised on the

experiments and improved the paper writing. All the authors had approved the final version.

ACKNOWLEDGMENT

We are very grateful to our laboratory members for fruitful discussions of advancing this research. We would also like to thank to all the students participated in the experiments.

REFERENCES

[1] Programming language. [Online]. Available: https://www.spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020

[2] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano, "A Java programming learning assistant system using test-driven development method," *IAENG Int. J. Comput. Sci*., vol. 40, no.1, pp. 38-46, Feb. 2013.

[3] K. K. Zaw, N. Funabiki, and W.-C. Kao, "A proposal of value trace problem for algorithm code reading in Java programming learning assistant system," *Inf. Eng. Express*, vol. 1, no. 3, pp. 9-18, Sep. 2015.

[4] N. Funabiki, Tana, K. K. Zaw, N. Ishihara, and W.-C. Kao, "A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system," *IAENG Int. J. Comput. Sci*., vol. 44, no. 2, pp. 247-260, May 2017.

[5] A. Kashihara, A. Terai, and J. Toyota, "Making fill-in-blank program problems for learning algorithm," in *Proc. Int. Conf. Comput. Edu*., pp. 776-783, 1999.

[6] K. Terada and Y. Watanobe, "Automatic generation of fill-in-the-blank programming problems," in *Proc. IEEE Int. Symp. Embed. Mult./Many-core Sys.-on-Chip (MCSoC)*, pp. 187-193, 2019.

[7] P. Brusilovsky and S. Sosnovsky, "Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK," *J. Edu. Res. Comput.*, vol. 5, no. 6, Sept. 2005.

[8] T. Kakeshita and M. Murata, "Application of programming education support tool pgtracer for homework assignment," *Int. J. Learn. Tech. Learn. Environ*., vol. 1, no. 1, pp. 41-60, 2018.

[9] L. N. Barros, A. P. S. Mota, K. V. Delgado, and P. M. Matsumoto, "A tool for programming learning with pedagogical patterns," in *Proc. OOPSLA Work. Eclipse Tech. Exchange*, pp. 125-129, Oct. 2005.

[10] CUP. [Online]. Available: http://czt.sourceforge.net/dev/java-cup/manual.html

[11] M. Banahan, D. Brady, and M. Doran, *The C Book*, 2nd ed., GBdirect, 1991.

[12] Codebind. [Online]. Available: http://www.codebind.com/c-examples

[13] Learn DS & Algorithms. [Online]. Available: https://www.programiz.com/dsa

[14] Pascal's Triangle. [Online]. Available: https://brilliant.org/wiki/pascals-triangle/

Copyright © 2021 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).

Htoo Htoo Sandi Kyaw received the B. E. and M. E. degrees in information science and technology from University of Technology (Yatanarpon Cyber City), Myamar, in 2015 and 2018, respectively. She is currently a Ph.D. candidate in Graduate School of Natural Science and Technology at Okayama University, Japan. Her research interests include educational technology and web application systems. She is a member of IEICE.

Nobuo Funabiki received the B.S. and Ph.D. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1984 and 1993, respectively. He received the M.S. degree in electrical engineering from Case Western Reserve University, USA, in 1991. From 1984 to 1994, he was with Sumitomo Metal Industries, Ltd., Japan. In 1994, he joined the Department of Information and Computer Sciences at Osaka University, Japan, as an assistant professor, and became an associate professor in 1995. He stayed at University of Illinois, Urbana-Champaign, in 1998, and at University of California, Santa Barbara, in 2000-2001, as a visiting researcher. In 2001, he moved to the Department of Communication Network Engineering (currently, Department of Electrical and Communication Engineering) at Okayama University as a professor. His research interests include computer networks, optimization algorithms, educational technology, and Web technology. He is a member of IEEE, IEICE, and IPSJ.

Shune Lae Aung received the B.S. degree in computer science from University of Yadanabon, Mandalay, Myanmar, in 2012, and the M.S. degree in computer science from University of Yangon, Myanmar, in 2015. In 2017, she joined Department of Computer Studies at University of Yangon, Myanmar, as a lecturer, where currently, she is also a Ph.D. candidate. Her research interests include educational technology, assistive technology, and human computer interaction.

Nem Khan Dim received the B.S. and M.S. degrees in computer science from University of Yangon, Myanmar, in 2008 and 2011, and Ph.D. in computer science from Kochi University of Technology, Japan, in 2016, respectively. She is currently a lecturer in Department of Computer Studies at University of Yangon, Myanmar. Her research interests include human-computer interaction and assistive technology.

Wen-Chung Kao received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taiwan, in 1992 and 1996, respectively. From 1996 to 2000, he was a Department Manager at SoC Technology Center, ERSO, ITRI, Taiwan. From 2000 to 2004, he was an Assistant Vice President at NuCam Corporation in Foxlink Group, Taiwan. Since 2004, he has been with National Taiwan Normal University, Taipei, Taiwan, where he is currently a Professor at Department of Electrical Engineering and the Dean of School of Continuing Education. His current research interests include system-on-a-chip (SoC), flexible electrophoretic display, machine vision system, digital camera system, and color imaging science. He is a senior member of IEEE.